SYSTEM AND METHOD FOR RECOVERING FROM MEMORY FAILURES IN COMPUTER SYSTEMS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to systems and methods for recoverable programming, and more particularly to a recoverable programming system and method for memory system failures in multi-processor computer systems.

2. Discussion of Background Art

Demand for increased performance and high availability of commodity computers is increasing with the ubiquitous use of computers and the Internet services which serve them. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief that software (SW) errors and administration time are, and will continue to be, the most probable cause of the loss of availability. While such failures are clearly commonplace, especially in desktop environments, it is believed that certain other hardware (HW) errors are also becoming more probable.

Processors, caches, and memories are becoming larger, faster and more dense, while being increasingly used in ubiquitous and adverse environments such as at high altitudes, in space, and in industrial applications. Articles, such as Ziegler, J. F., et al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)", IBM Journal of R&D, vol 40, no 1, pp 3-18, January 1996, and Ziegler, J. F., "Terrestrial Cosmic Rays", IBM Journal of R&D, vol 40, no 1, pp 19-40, January 1996, have shown that these changes will lead to increased transient errors in CMOS memory due to the effects of

cosmic rays, approximately 6000 FIT (1 FIT equals 1 failure in 10 9 h) for one 4Mbit DRAM.

Tandem (see, Compaq Corporation, "Data Integrity for Compaq NonStop Himalaya Servers", White Paper, 1999) indicates that such errors also apply to processor cores or on-chip caches at modern die sizes/voltage levels. They claim that processors, cache, and main memory are all susceptible to high transient error rates. A typical processor's silicon can have a soft-error rate of 4000 FIT, of which approximately 50% will affect processor logic and 50% the large on-chip cache. Due to increasing speeds, denser technology, and lower voltages, such errors are likely to become more probable than other single hardware component failures. With the increasing evolution to larger tightly interconnected commodity machines (such as Sun's Enterprise 10000 machines), the probability of soft-errors and error containment problems increases further. Soft-error probability increases not only due to increased system scale, but also due to an increased number of components on the memory access path. Since the machines are tightly coupled, memory path soft-errors introduce error containment problems which without some form of soft-ware error containment can lead to complete loss of availability.

Techniques such as Error Correction Codes (ECC) and ChipKill (see, Dell, T. J., "A White Paper on the benefits of Chipkill Correct ECC for PC Server Main Memory", IBM Microelectronics Division, Nov. 1997) have been used in main memories and interconnects to correct some of these errors (90% for ECC). Unfortunately such techniques, only help reduce visible error rates for semiconductor elements that can be covered by such codes (large storage elements). With raw error rates increasing with technological progress and more complicated interconnected memory subsystems, ECC is unable to address all the soft-error problems. For example, a 1Gb memory system based

on 64Mbit DRAMs still has a combined visible error rate of 3435 FIT when using Single Error Correct Double Error Detect (SECDED) ECC. This is equivalent to around 900 errors in 10000 machines in 3 years. Unfortunately, current commodity hardware and software provide little to no support for recovery from errors not covered by ECC whether detected or not. Such problems have been considered by mainframe technology for years, but in the field of commodity hardware, it is currently not cost effective to provide full redundancy/support in order to mask errors. Therefore, the burden falls to commodity hardware and the software using it to attempt to handle these errors for the highest availability.

Most contemporary commodity computer systems, while providing good performance, pay little attention to availability issues resulting from such errors. For example, the IA-32 architecture supports only ECC on main memory rather than across the system, requiring system reboot on errors not covered by this ECC. Consequently, commodity software such as the OS, middleware and applications have been unable to deal with the problem. Future commodity processor architectures may provide support to detect and notify the system of such probable errors. For instance, upcoming IA-64 processors, while not recoverable in the general case, do offer some support with certain limitations.

Availability in computer systems is determined by hardware and software reliability. Hardware reliability has traditionally existed only in proprietary servers, with specialized redundantly configured hardware and critical software components, possibly with support for processor pairs (see, Bartlett, J., "A Nonstop Kernel", Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar, Ca, pp 22-29, Dec. 1981), e.g. IBM S/390 Parallel Sysplex (see, Nick, J.M., et al., "S/390 Cluster

Technology: Parallel Sysplex", *IBMSystems Journal,* vol 36, no 2., pp 172-201, 1997), and Tandem NonStop Himalaya (see, Compaq, Product description for Tandem Nonstop Kernel 3.0. Download Feb. 2000, http://www.tandem.com).

Sysplex supports hot swap execution, redundant shared disk with fault-aware system software for error detection and fail-over restart. Tandem supports redundant fail-over lock-stepped processors with a NonStop kernel and middleware, which provide improved integrity through the software stack. These systems provide full automatic support to mask the effects of data and resource loss. They rely on reliable memory and fail-over rather than direct memory error recovery. Another approach is fault containment and recovery with "node" granularity. In these systems, each node has its own kernel. When one node fails, the others can recover and continue to provide services. Systems of this type include the early cluster systems (see, Pfister, G., "In Search of Clusters", Prentice Hall, 1998), and NUMA architectures, such as Hive (see, Chapin, J., et al., "Hive: Fault Containment for Shared Memory Multiprocessors," *Proc. of the 15th SOSP*, Dec.1995, pp 12-25, and Teodosiu, D., et al., "Hardware Fault Containment in Scalable Shared Memory Multiprocessors," *Proc. of the 24th ISCA*, pp 73-84, June 1997).

Hardware faults are difficult to catch and repeat.

Software reliability has been more difficult to achieve in commodity software even with extensive testing and quality assurance. Commodity software fault recovery has not evolved very far. Most operating systems support some form of memory protection between units of execution to detect and prevent wild read/writes. But most commodity operating systems have not tackled problems of memory errors themselves or taken up software reliability research in general. Examples include Windows 2000 and Linux. They typically rely on failover solutions, such as Wolfpack by Microsoft. A lot of work

has been undertaken in the fault-tolerant community regarding the problems of reliability and its recovery in software (see, Brown, N.S. and Pradhan, D.K. "Processor and Memory-Based Checkpoint And Rollback Recovery", IEEE Computer, pp 22-31, Feb. 1993; Gray, J., and Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan Kaufmann, 1993; and Kermarrec, AM., et al., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", *Proc. of the 25th FTCS*, pp 289-298, June 1995).

These include techniques such as checkpointing and backward error recovery. A lot of this work has been conducted in the context of distributed systems rather than in single systems. There are also techniques for efficient recoverable software components, e.g. RIO file cache (see, Chen, P.M., et al., "The Rio File Cache: Surviving Operating System Crashes", *Proc. of the 7th ASPLOS,* pp 74-83, October 1996), and Recoverable Virtual Memory (RVM) (see, Satyanarayanan, et al. "Lightweight Recoverable Virtual Memory". Proc. SOSP, pp 146-160, Dec. 1993).

Rio takes an interesting software-based approach to fault containment aimed at a fault-tolerant file cache, but with general uses. By instrumenting access to shared data structures with memory protection operations, wild access to the shared data structures becomes improbable.

Other methods for handling memory errors include a try-except block solution. In general, the try-except mechanism itself is not sufficient to handle memory failures. The saved state needed for memory failures is more extensive (as an example, for IA-64 architecture) than what can be obtained by try-except. Thus saving state is an expensive operation in terms of system overhead.

Since current responses to memory failures are costly to invoke and execute, do not guarantee recovery under all cases for next generation processors, such as IA64, and are impossible to recover at all for current generations of commodity processors, such as the IA32 family, what is needed is a system and method for recoverable programming that overcomes the problems of the prior art.

SUMMARY OF THE INVENTION

The present invention is a system and method for recovering from memory failures in computer systems. The method of the present invention includes the steps of: identifying a predetermined instruction sequence; monitoring for memory access errors in response to the request; logging a memory access error in an error logging register; polling the register for any logged memory access error during execution of the instruction sequence; and raising exceptions, if the memory access error is logged.

In other aspects of the invention, the method may include the steps of: checkpointing a predetermined set of system data; recovering from the memory access error using the checkpointed system data, if the memory access error is logged during execution of the instruction sequence; setting data returned in response to the memory access request equal to a set of predefined fake data, if the memory access error is logged during execution of the instruction sequence; skipping the polling and raising steps if the data returned in response to the memory access request is not equivalent to the predefined fake data; masking a machine check abort handle; updating pointers, if the memory access error is logged; and re-executing the memory access request, if software so commands.

Within the system of the present invention, memory access errors are stored in an error logging register, hardware generated machine check abort handles are masked, and a memory controller, which responds to memory access errors is under full control of either O/S or application software so that memory access errors are intercepted and responded to without necessitating a system reboot.

The system and method of the present invention are particularly advantageous over the prior art because memory failures in commodity multiprocessor computers can

now be responded to and remedied without rebooting the computer. The present invention will succeed in responding to memory errors much more effectively than standard machine check abort handles.

The present invention is particularly applicable to O/S level code which can not otherwise be restarted in response to memory errors without rebooting. When the present invention is incorporated within application level code, the present invention also enables the application to recover from memory errors, instead of otherwise being shut down and restarted.

These and other aspects of the invention will be recognized by those skilled in the art upon review of the detailed description, drawings, and claims set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a dataflow diagram of a system for recovering from memory access failures;

Figure 2 is a flowchart of a first embodiment of a method for recovering from memory access failures;

Figure 3 is a flowchart of a second embodiment of the method for recovering from memory access failures; and

Figure 4 is a flowchart of a third embodiment of the method for recovering from memory access failures.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 1 is a dataflow diagram of a system 100 for recovering from memory access failures. The system 100 includes a memory 102, a memory controller 104, error logging registers 106, and a central processing unit (CPU) 108, coupled together by a bus 110. The CPU 108 is controlled by software 112. The software 112 is preferably included within the system's 100 operating system, however the software 112 could also be instantiated within an application program as well. The software 112 configures the memory controller 104 and has access to the error registers 106. Operation of the system 100 is discussed with respect to method Figures 2, 3, and 4.

Figure 2 is a flowchart of a first embodiment of a method 200 for recovering from memory access failures. The method 200 begins with step 202 where the software 112 identifies a predetermined critical computer instruction sequence about to be executed by the CPU 108, which includes a memory access request. The predetermined critical computer instruction sequence can be part of a set of instruction sequences, identified by the software 112 designer, for which error recovery is required. While the critical computer instruction sequence discussed below include a memory access request, those skilled in the art will know that concepts discussed with respect to recovery from a memory access request error can be applied to other recovery critical instruction sequences which would otherwise require rebooting of the system 100 to recover from. Thus, the present invention is particularly applicable to O/S level code which can not otherwise be restarted in response to memory errors without rebooting. When the present invention is incorporated within application level code, the present invention enables the application to recover from memory errors, instead of otherwise being shut down and

restarted. The present invention may also be used on non-critical computer instruction sequences and for non-memory related errors.

In step 204, the software 112 then instructs the memory controller 104 to mask any raised machine check abort (MCA) handle. Next in step 206, the CPU 108 executes the memory access request. The memory controller 104 logs any memory access error in the error logging register 106, in step 208. Next, in step 210, the software 112 polls the error logging register 106 for any memory access errors, during execution of the instruction sequence.

In step 212, the software 112 raises exceptions and updates pointers, if a memory access error was logged during execution of the instruction sequence. The exceptions perform various diagnostic functions in response to the memory error. The housekeeping functions may include system recovery, memory management, and other reset procedures. Pointers are updated when during memory error diagnosis, there are indications that a portion or sector of the memory 102 may be physically damaged or corrupt.

Depending upon the memory access error which occurred, the software 112 may command the CPU 108 to re-execute the memory access request, in step 214. The software 112 will command the CPU 108 to re-execute the memory access request if the memory access error detected is most likely due to a transitory error condition, which is not likely to occur again. On the other hand, if the memory access error suggest that the memory 102 itself is physically damaged, the software 112 will not instruct the CPU 108 to re-execute the memory access request. In step 216, the software 112 instructs the memory controller 104 to enable the MCA handle.

Figure 3 is a flowchart of a second embodiment of the method for recovering from memory access failures. The method 300 begins with step 302 where the software 112 identifies a predetermined critical computer instruction sequence about to be executed by the CPU 108, which includes a memory access request.

In step 304, the software 112 then checkpoints a predetermined set of system data necessary to recover should the memory access request fail. Checkpointing is component of a transactional paradigm in which permanent modifications to system data are not made until all associated operations within the transaction have been successfully committed. Thus if during a transaction, such as the memory access request, an error occurs, the system data stored during the checkpoint can be restored.

In step 306, the software 112 then instructs the memory controller 104 to mask any raised machine check abort (MCA) handle. In step 308, the CPU 108 executes the memory access request. The memory controller 104 logs any memory access error in the error logging register 106, in step 310. Next, in step 312, the software 112 polls the error logging register 106 for any memory access errors, during execution of the instruction sequence. If a memory access error is logged during execution of the instruction sequence, the software 112: raises exceptions and updates pointers, in step 314; recovers the checkpointed system data, in step 316; and restores the system data, in step 318

As discussed above, with reference to Figure 2, depending upon the memory access error which occurred, the software 112 may command the CPU 108 to re-execute the memory access request, in step 320. In step 322, the software 112 instructs the memory controller 104 to enable the MCA handle.

Figure 4 is a flowchart of a third embodiment of the method for recovering from memory access failures. The method 400 begins with step 402 where the software 112 identifies a predetermined critical computer instruction sequence about to be executed by the CPU 108, which includes a memory access request. In step 404, the software 112 then instructs the memory controller 104 to mask any raised machine check abort (MCA) handle. In step 406, the CPU 108 executes the memory access request. The memory controller 104 logs any memory access error in the error logging register 106, in step 408.

In step 410, the memory controller 104 sets data returned in response to the memory access request equal to a set of predefined fake data, if a memory access error is logged during execution of the instruction sequence. The software 112 has preprogrammed the memory controller 104 to perform the functionality described in step 410. By setting the returned data to the predefined fake data in when a memory access error occurs, corrupted data is not returned to the software, which might otherwise necessitate a system reboot.

In step 412, the software 112 receives data returned in response to the memory access request. In step 414, the method 400 skips to step 422, if the data returned in response to the memory access request is not equivalent to the predefined fake data. When the data returned is not equal to the fake data, the software 112 knows that no memory access error has occurred, during execution of the instruction sequence, even though the software 112 has not polled the error logging register. Thus, the polling step can be eliminated, speeding up the memory access request.

In step 416, the software 112 polls the error logging register 106 for any memory access errors, during execution of the instruction sequence. In step 418, the software 112

raises exceptions and updates pointers, if a memory access error was logged during execution of the instruction sequence.

As discussed above, with reference to Figure 2, depending upon the memory access error which occurred, the software 112 may command the CPU 108 to re-execute the memory access request, in step 420. In step 422, the software 112 instructs the memory controller 104 to enable any hardware raised MCA handles.

Another enhancement which may be applied to each of the three embodiments discussed above, is to batch access to memory in large chunks whenever possible. By batch accessing data, memory access errors are logged and polled for the entire batch. This has implication on a granularity of the system 100 operation and is limited by pointer manipulation.

While one or more embodiments of the present invention have been described, those skilled in the art will recognize that various modifications may be made. Variations upon and modifications to these embodiments are provided by the present invention, which is limited only by the following claims.